Approaches to Solving the Travelling Salesman Problem: Ant-Colony Optimization and Genetic Algorithms

João Pinheiro - up202008133 João Oliveira - up202004407
Ricardo Cavalheiro - up202005103

EDAA 23/24

Abstract

The Travelling Salesman Problem (TSP) is a classic problem in combinatorial optimization, notable for its applications and complexity. This report explores two approaches for solving the TSP: Ant-Colony Optimization (ACO) and Genetic Algorithms (GA), including the parallel implementation of ACO. We explore the principles behind each method, their implementation details, their performance on different problem sizes, and their respective strengths and weaknesses.

1 Introduction

The Travelling Salesman Problem (TSP) is defined as finding the shortest possible route that visits each city exactly once and returns to the origin city. This problem is a quintessential NP-hard problem, meaning that there is no known polynomial-time solution that solves all instances of TSP efficiently. The TSP has extensive applications in logistics, manufacturing, and DNA sequencing, making effective heuristic solutions highly valuable.

2 The Travelling Salesman Problem

The TSP can be formally described as follows: given a set of n cities and the distances between each pair of cities, find the shortest possible route that visits each city once and returns to the starting point. Mathematically, it can be formulated as finding the minimum weight Hamiltonian cycle in a weighted, undirected graph.

2.1 NP-Hard Problems

TSP is categorized as NP-hard, indicating that as the number of cities increases, the number of possible tours grows factorially. For example, a problem with 10 cities has $10! = 3,628,800$ possible permutations, making exhaustive search impractical even for relatively small instances. NP-hard problems require heuristic or approximation methods for large instances to find near-optimal solutions in a reasonable time.

2.2 Naive Approach

The naive approach to solving the TSP involves evaluating all possible permutations of city visits to find the shortest tour. While straightforward, this brute-force method is computationally infeasible for all but the smallest instances due to its factorial time complexity. Specifically, the number of possible tours increases factorially with the number of cities, denoted as $n!$. For instance, a problem with just 10 cities has $10! = 3,628,800$ possible permutations. As the number of cities grows, the number of permutations grows extremely rapidly (e.g., 20 cities result in $20! \approx 2.43 \times 10^{18}$ permutations). This exponential growth means that even with modern computational power, exhaustively searching through all possible routes becomes impractical for larger instances, requiring heuristic or approximation methods to find near-optimal solutions in a reasonable time.

3 Ant-Colony Optimization Approach

Ant-Colony Optimization (ACO) is a nature-inspired algorithm that mimics the foraging behavior of ants. Ants communicate and find shortest paths via pheromone trails, a mechanism that ACO leverages to solve optimization problems like the TSP.

3.1 ACO Terminology

To understand the Ant-Colony Optimization (ACO) algorithm, it is crucial to grasp the terminology and core concepts that underpin its functionality. These concepts are derived from the foraging behavior of real ants and adapted into a computational framework for solving optimization problems like the Travelling Salesman Problem (TSP).

3.1.1 Pheromone Trails

In ACO, pheromone trails represent the virtual paths that artificial ants follow. These trails are crucial as they encode the learned desirability of taking specific routes between cities. Pheromone levels are initialized uniformly but dynamically updated based on the quality of solutions found by the ants. Higher pheromone levels indicate more promising paths, thereby guiding future ants more effectively. The concentration of pheromone on an edge is denoted by τ_{ij}, where i and j are two cities.

3.1.2 Heuristic Information

Heuristic information is typically the inverse of the distance between two cities, denoted as $\eta_{ij} = \frac{1}{d_{ij}}$, where d_{ij} is the distance between city i and city j. This heuristic encourages ants to choose shorter paths by increasing the attractiveness of edges with smaller distances.

3.1.3 Path Construction

Ants build solutions step by step, starting from a randomly chosen city and moving to subsequent cities based on a probabilistic decision rule. The probability P_{ij} that an ant at city i will move to city j is influenced by both the pheromone trail τ_{ij} and the heuristic information η_{ij}, as described by the formula:

$$P_{ij} = \frac{\tau_{ij}^{\alpha} \eta_{ij}^{\beta}}{\sum_{k \in N_i} \tau_{ik}^{\alpha} \eta_{ik}^{\beta}}$$

where N_i is the set of cities not yet visited by the ant, α controls the influence of the pheromone trail, and β controls the influence of the heuristic information.

3.1.4 Pheromone Update

After all ants have completed their tours, pheromone trails are updated to reflect the paths taken and their respective quality. This update process has two components: pheromone evaporation and pheromone deposit. Evaporation decreases the pheromone level on all edges by a factor of $(1 - \rho)$, where ρ is the evaporation rate. This prevents the algorithm from prematurely converging to suboptimal solutions by reducing the influence of older paths. The pheromone deposit increases the pheromone levels on the edges that are part of good solutions. The amount of pheromone $\Delta\tau_{ij}$ deposited on an edge (i, j) is typically inversely proportional to the length of the tour in which the edge was included.

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

where m is the number of ants and $\Delta\tau_{ij}^{k}$ is the pheromone deposited by the k-th ant.

By balancing pheromone evaporation and deposition, the algorithm encourages exploration of new paths while exploiting known good solutions, thus navigating the trade-off between exploration and exploitation effectively.

3.2 ACO Algorithm

Algorithm 1 ACO Algorithm

```
1: Initialize pheromone levels $\tau_{ij}$ for all edges $(i, j)$
2: Set parameters $\alpha$ (pheromone importance), $\beta$ (heuristic importance), and $\rho$
   (evaporation rate)
3: for each iteration do
4:    for each ant do
5:       Randomly place the ant on a starting city
6:       while the ant has not visited all cities do
7:          Select the next city $j$ based on transition probability:
                $$P_{ij} = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{k \notin \text{visited}} \tau_{ik}^{\alpha}\eta_{ik}^{\beta}}$$
            where $\eta_{ij} = \frac{1}{d_{ij}}$ is the heuristic information.
8:          Move to city $j$
9:       end while
10:      Complete the tour and calculate its length
11:   end for
12:   Update pheromone levels:
                $$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum \Delta\tau_{ij}$$
      where $\Delta\tau_{ij}$ depends on the quality of the tours.
13: end for
```

3.3 Parallelization in ACO

ACO can be parallelized due to the independence of artificial ants. Each ant explores the solution space autonomously, relying on local information and pheromone trails. They don't need direct coordination, making parallelization simple. By distributing workload across multiple processors, ACO can handle larger problems efficiently, speeding up the search for optimal solutions.

3.4 Ant Colony Optimization Discussion

Ant Colony Optimization (ACO) is a robust and nature-inspired algorithm that excels at solving complex optimization problems, particularly those involving pathfinding and combinatorial optimization. ACO is inspired by the foraging behavior of ants, which use pheromone trails to communicate and find the shortest paths between their colony and food sources. However, the performance of an ACO algorithm heavily depends on the specific details of its implementation, particularly the design of pheromone update rules and the balance between exploration and exploitation.

The pheromone update rule is a critical component of ACO. It dictates how pheromone levels are increased on paths that are frequently used by ants and how they evaporate over time. A well-designed pheromone update rule ensures that the algorithm effectively reinforces good solutions while avoiding premature convergence to suboptimal paths. If the update rule is too aggressive, the algorithm may converge too quickly, missing out on potentially better solutions. Conversely, if it is too lenient, the algorithm might wander excessively, failing to efficiently hone in on the optimal paths.

Balancing exploration and exploitation is another key aspect. Exploration refers to the ability of the ants to search new paths and discover unexplored areas of the solution space, while exploitation involves refining and optimizing known good paths. Achieving the right balance is crucial: too much exploration can lead to inefficiency and excessive computation time, whereas too much exploitation can cause the algorithm to get stuck in local optima. Strategies such as introducing random decisions or dynamically adjusting pheromone levels can help maintain this balance.

Several other parameters influence the effectiveness of ACO, including the number of ants, the pheromone evaporation rate, and heuristic factors that guide the ants' path selection. Fine-tuning these parameters requires a good understanding of the specific problem domain and often involves extensive experimentation.

Moreover, parallelization can significantly enhance ACO's performance. By distributing computational tasks across multiple processors or threads, parallelization accelerates the search process, enabling the algorithm to explore the solution space more efficiently. This can lead to faster convergence and improved solution quality, making parallel ACO a compelling choice for tackling large-scale optimization problems.

4 Genetic Algorithm Approach

Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random searches provided with historical data to direct the search into the region of better performance in solution space. Genetic algorithms simulate the process of natural selection which means those species that can adapt to changes in their environment can survive and reproduce and go to the next generation. In simple words, they simulate "survival of the fittest" among individuals of consecutive generations to solve a problem.

4.1 Genetic Algorithm Terminology

Before delving into the specifics of the algorithm itself, it's essential to familiarize ourselves with some important terms in the context of Genetic Algorithms. Understanding these concepts will lay a solid foundation and make it easier to comprehend the workings of the algorithm. Here are some key terms:

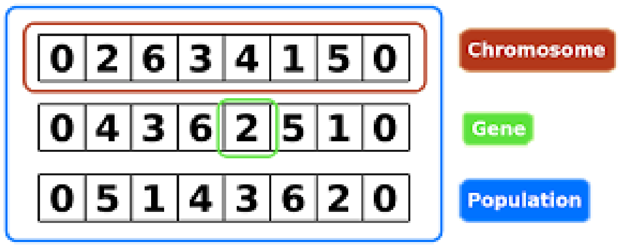

Figure 1: Chromosome, Gene & Population

- **Gene**: A gene is a basic unit of information within a chromosome. It represents a specific characteristic or part of the solution.
- **Chromosome**: A chromosome represents a single solution within the population. It is typically encoded as a string of genes, which can be bits, numbers, or other data structures. In our case, it is a path.
- **Population**: A population refers to a collection of feasible solutions to the problem at hand. Each individual within this population is known as a chromosome.

- **Mutation Function**: It's a genetic operator that introduces random changes to a chromosome's genes. This helps maintain genetic diversity within the population and allows the algorithm to explore new solutions.

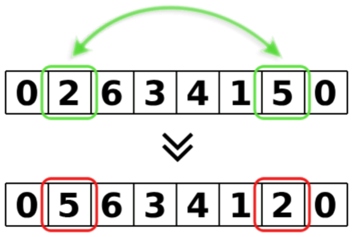

Figure 2: Example of mutation function

- **Crossover**: It's a genetic operator used to combine the genetic information of two parent chromosomes to produce one or more offspring. This process mimics biological reproduction and aims to create new, potentially better solutions.

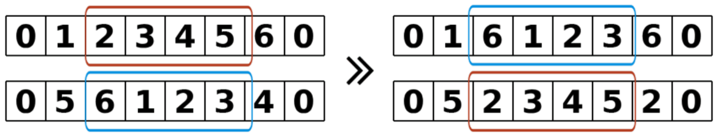

Figure 3: Example of crossover function

- **Tournament Selection**: It's a method used in genetic algorithms to choose individuals from the population for reproduction. It works by randomly selecting a subset of individuals from the population, known as the tournament size. These individuals compete against each other based on their fitness values. The individual with the highest fitness in this subset is chosen as a parent for reproduction. This process is repeated until the required number of parents is selected.

- **Roulette Wheel Selection**: Also known as fitness proportionate selection, is a technique used to select individuals for reproduction in genetic algorithms. It assigns selection probabilities to individuals based on their fitness values. Each individual's probability of being selected is proportional to their fitness. The process can be visualized as a roulette wheel where each individual occupies a segment proportional to their fitness. The wheel is spun to randomly select individuals, giving those with higher fitness a greater chance of being chosen.

4.2 Genetic Algorithm Pseudo-Code

Algorithm 2 GA Algorithm

```
1: Initialize population of chromosomes
2: for each generation do
3:     Evaluate fitness of each chromosome based on the tour length
4:     Select parents using selection method (e.g., roulette wheel, tournament)
5:     Apply crossover to generate new offspring
6:     Apply mutation to offspring with a certain probability
7:     Form a new population by selecting the best chromosomes from parents
       and offspring
8: end for
```

4.3 Genetic Algorithm Discussion

Genetic algorithms (GAs) are a versatile and powerful class of optimization techniques inspired by the principles of natural selection and genetics. They are particularly useful for solving complex problems where traditional methods may falter, such as those involving large search spaces or intricate, nonlinear relationships. However, the efficacy of a genetic algorithm is highly contingent upon its implementation specifics, particularly the design of crossover and mutation functions.

The crossover function, which combines the genetic information of parent solutions to produce offspring, plays a crucial role in exploring the solution space. A well-designed crossover function ensures that useful traits are inherited and recombined effectively, fostering the discovery of optimal solutions. On the other hand, a poorly designed crossover function can lead to premature convergence or fail to adequately explore the solution space, thereby limiting the algorithm's performance.

Similarly, the mutation function introduces variability by randomly altering some aspects of the offspring. This variability is essential for maintaining genetic diversity within the population, which helps prevent the algorithm from becoming trapped in local optima. The balance between exploration (searching new areas) and exploitation (refining known good areas) is delicate; an overly aggressive mutation rate can disrupt good solutions, while a too-conservative rate may lead to stagnation.

Other parameters such as population size, selection mechanisms, and termination criteria also significantly influence the performance of genetic algorithms. Fine-tuning these parameters to the specific problem at hand is often more of an art than a science and requires careful experimentation and domain knowledge.

The success of a genetic algorithm depends not just on the implementation of these functions but on how well they are tailored to the particularities of the problem being solved.

5 Results and Discussion

In this section, we present and analyze the results obtained from implementing the Ant-Colony Optimization (ACO) and Genetic Algorithm (GA) approaches to solving the Travelling Salesman Problem (TSP). We compare the performance of both methods in terms of computational efficiency and solution quality across various problem sizes. Additionally, we discuss the impact of parallelization on ACO's performance and highlight the strengths and weaknesses of each approach. Through detailed analysis, we aim to provide insights into the practical applications and limitations of ACO and GA for TSP, guiding future research and implementation efforts in combinatorial optimization.

5.1 ACO Serial vs Parallel Performance

Although there is an absolute speed increase, the significant cost of node-to-node transmission makes this not so beneficial. The communication time between nodes rises with node count, which results in a drop in speedup relative to ideal speedup. It's worth noting that beyond four cores, the speed-up is negligible.

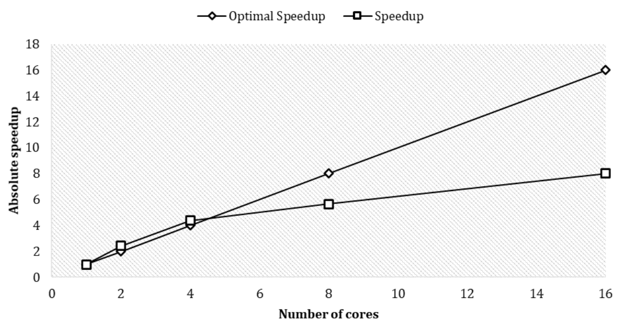

Figure 4: ACO Serial & Parallel Performance

5.2 Performance Comparison

As evidenced by the results, the ant colony approach exhibits poorer performance when dealing with a small number of cities. However, as the number of cities increases, its efficiency surpasses that of the genetic algorithm significantly.

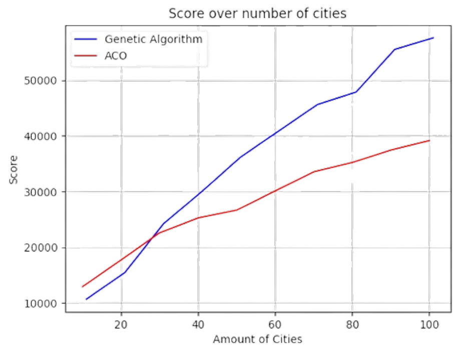

Figure 5: Score over nr. of cities - ACO vs GA

The ant colony approach exhibits superior temporal efficiency irrespective of input scale, outperforming alternative methodologies such as the genetic algorithm in computational speed and resource allocation, consistently across various dataset sizes.

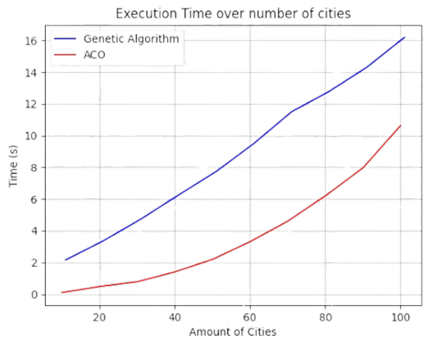

Figure 6: Execution time over nr. of cities - ACO vs GA

5.3 Advantages and Disadvantages

Ant-Colony Optimization offers several advantages, including an effective balance between exploration and exploitation, inherent parallelism and scalability, as well as adaptability to dynamic environments and various problem constraints. However, it also has its drawbacks, being sensitive to parameter settings, challenging for theoretical analysis, and having an uncertain convergence time. On the other hand, Genetic Algorithm presents advantages such as versatility and easy implementation, effectiveness in finding near-optimal solutions across a broad spectrum of problems, and strong performance in smaller instances. Nonetheless, it suffers from disadvantages like computational expense for larger problems, tendency for early convergence to suboptimal solutions, and sensitivity to the choice of genetic operators and parameter settings.

6 Conclusions

Based on our findings, in general, the Ant-Colony Optimization (ACO) approach exhibited superior performance compared to the Genetic Algorithm (GA) method. However, it's noteworthy that the GA did yield better outcomes when confronted with a smaller number of cities, yet its efficacy waned when dealing with larger inputs. While this suggests a potential superiority of the ACO approach over GA, drawing definitive conclusions necessitates a nuanced consideration of several factors. Indeed, the superiority of one method over the other is contingent upon various variables, including the complexity of the problem at hand, the size of the input data, specific requirements or constraints imposed by the task, and even the coding methodology employed. Hence, while the ACO approach might prove more effective for this particular problem scenario, the optimal choice between ACO and GA hinges on a careful evaluation of these multifaceted factors.

7 References

- Manning Publications. *Grokking Artificial Intelligence Algorithms.* Available at: `https://www.manning.com/books/grokking-artificial-intelligence-algorithms`
- ScienceDirect. *LPWAN Technologies for IoT and M2M Applications.* Available at: `https://www.sciencedirect.com/book/9780128188804/lpwan-technologies-for-iot-and-m2m-applications`
- Kumar, D. N., & Reddy, M. J. (2006). *Ant Colony Optimization for Multi-Purpose Reservoir Operation.* Water Resources Management.
- Dorigo, M., Birattari, M., & Stützle, T. (2006). *Ant Colony Optimization – Artificial Ants as a Computational Intelligence Technique.* IEEE Computational Intelligence Magazine.
- Katiyar, S., Nasiruddin, I., & Ansari, A. Q. (2015). *Ant Colony Optimization: A Tutorial Review.*